■矩形クラス

```
// BoundingRectangle.h
#ifndef BOUNDING_RECTANGLE_H_
#define BOUNDING_RECTANGLE_H_

#include <gslib.h>

// 矩形クラス
class BoundingRectangle {
public:
    // コンストラクタ
    BoundingRectangle() = default;
    // コンストラクタ
    BoundingRectangle(float left, float top, float right, float bottom);
    // コンストラクタ
    BoundingRectangle(const GSvector2& min, const GSvector2& max);
    // 点が矩形の内側に入っているか?
    bool containts(const GSvector2& position) const;
    // 矩形が重なっているか
    bool intersects(const BoundingRectangle& other) const;
    // 平行移動
    BoundingRectangle translate(const GSvector2& position) const;
    // サイズの拡張
    BoundingRectangle expand(const GSvector2& size) const;
    // 幅
    float width() const;
    // 高さ
    float height() const;
    // 左上の座標
    const GSvector2& min() const;
    // 右下の座標
    const GSvector2& max() const;

private:
    GSvector2 min_{ 0.0f, 0.0f };     // 矩形の左上の座標
    GSvector2 max_{ 0.0f, 0.0f };     // 矩形の右下の座標
};

#endif
```

```cpp
// BoundingRectangle.cpp
#include "BoundingRectangle.h"

// コンストラクタ
BoundingRectangle::BoundingRectangle(float left, float top, float right, float bottom)
    : BoundingRectangle{ { left, top }, { right, bottom } } {
}

// コンストラクタ
BoundingRectangle::BoundingRectangle(const GSvector2& min, const GSvector2& max)
    : min_{ min }, max_{ max } {
}

// 点が矩形の内側に入っているか?
bool BoundingRectangle::containts(const GSvector2& position) const {
    return (min_.x <= position.x && position.x <= max_.x)
        && (min_.y <= position.y && position.y <= max_.y);
}

// 矩形が重なっているか
bool BoundingRectangle::intersects(const BoundingRectangle & other) const {
    if (min_.x > other.max_.x) return false;
    if (max_.x < other.min_.x) return false;
    if (min_.y > other.max_.y) return false;
    if (max_.y < other.min_.y) return false;
    return true;
}


// 平行移動
BoundingRectangle BoundingRectangle::translate(const GSvector2& position) const {
    return{ min_ + position, max_ + position };
}

// サイズの拡張
BoundingRectangle BoundingRectangle::expand(const GSvector2& size) const {
    return{ min_ - size, max_ + size };
}

// 幅
float BoundingRectangle::width() const {
    return max_.x - min_.x;
}

// 高さ
float BoundingRectangle::height() const {
    return max_.y - min_.y;
}

// 左上の座標
const GSvector2& BoundingRectangle::min() const {
    return min_;
}

// 右下の座標
const GSvector2& BoundingRectangle::max() const {
    return max_;
}
```

■アクタークラス

```
// Actor.h
#ifndef ACTOR_H_
#define ACTOR_H_

#include <gslib.h>
#include <string>
#include "BoundingRectangle.h"

class IWorld;
enum class EventMessage;

// アクタークラス
class Actor {
public:
    // コンストラクタ
    Actor() = default;
    // 仮想デストラクタ
    virtual ~Actor() {}
    // 更新
    virtual void update(float delta_time);
    // 描画
    virtual void draw() const;
    // 衝突時リアクション
    virtual void react(Actor& other);
    // メッセージ処理
    virtual void handle_message(EventMessage message, void* param = nullptr);
    // 衝突判定
    void collide(Actor& other);
    // 死亡する
    void die();
    // 衝突判定しているか？
    bool is_collided(const Actor& other) const;
    // 死亡しているか？
    bool is_dead() const;
    // 名前を取得
    const std::string& name() const;
    // 座標を取得
    GSvector2 position() const;
    // 回転角度を取得
    float rotation() const;
    // 移動量を取得
    GSvector2 velocity() const;
    // 衝突判定データを取得
    BoundingRectangle body() const;
    // コピー禁止
    Actor(const Actor& other) = delete;
    Actor& operator = (const Actor& other) = delete;

protected:
    // ワールド
    IWorld*             world_{ nullptr };
    // 名前
    std::string         name_{};
    // 座標
    GSvector2           position_{ 0.0f, 0.0f };
    // 回転角度
    float               angle_{ 0.0f };
    // 移動量
    GSvector2           velocity_{ 0.0f, 0.0f };
    // 衝突判定
    BoundingRectangle   body_{ 0.0f, 0.0f, 0.0f, 0.0f };
    // 死亡フラグ
    bool                dead_{ false };
    // テクスチャＩＤ
    GSuint              texture_{ 0 };
};

#endif
```

```cpp
// Actor.cpp
#include "Actor.h"

// 更新
void Actor::update(float) {}

// 描画
void Actor::draw() const {
    gsDrawSprite2D(texture_, &position_, NULL, NULL, NULL, NULL, angle_);
}

// 衝突時リアクション
void Actor::react(Actor&) {}

// メッセージ処理
void Actor::handle_message(EventMessage, void*) {}

// 衝突判定
void Actor::collide(Actor& other) {
    if (is_collided(other)) {
        react(other);
        other.react(*this);
    }
}

// 死亡する
void Actor::die() {
    dead_ = true;
}

// 衝突判定
bool Actor::is_collided(const Actor& other) const {
    return body().intersects(other.body());
}

// 死亡しているか？
bool Actor::is_dead() const {
    return dead_;
}

// 名前を返す
const std::string& Actor::name() const {
    return name_;
}

// 座標を返す
GSvector2 Actor::position() const {
    return position_;
}

// 回転変換行列を返す
float Actor::rotation() const {
    return angle_;
}

// 移動量を取得
GSvector2 Actor::velocity() const {
    return velocity_;
}

// 衝突判定データを取得
BoundingRectangle Actor::body() const {
    return body_.translate(position_);
}
```

■アクターポインタクラス

```
// ActorPtr.h
#ifndef ACTOR_PTR_H_
#define ACTOR_PTR_H_

#include <memory>

// アクターポインタ
class Actor;
using ActorPtr = std::shared_ptr<Actor>;

// アクターの生成
template<class T,   class... Args>
inline ActorPtr new_actor(Args&&... args) {
    return std::make_shared<T>(args...);
}

#endif
```

■アクターマネージャークラス

```
// ActorManager.h
#ifndef ACTOR_MANAGER_H_
#define ACTOR_MANAGER_H_

#include "Actor.h"
#include "ActorPtr.h"
#include <list>

// アクターマネージャー
class ActorManager {
public:
    // コンストラクタ
    ActorManager() = default;
    // アクターの追加
    void add(const ActorPtr& actor);
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // 衝突判定
    void collide();
    // 衝突判定
    void collide(Actor& other);
    // 衝突判定
    void collide(ActorManager& other);
    // 削除
    void remove();
    // アクターの検索
    ActorPtr find(const std::string& name) const;
    // アクター数を返す
    unsigned int count() const;
    // メッセージ処理
    void handle_message(EventMessage message, void* param);
    // 消去
    void clear();
    // コピー禁止
    ActorManager(const ActorManager& other) = delete;
    ActorManager& operator = (const ActorManager& other) = delete;
private:
    using ActorList = std::list<ActorPtr>;
    // アクターリスト
    ActorList   actors_;
};

#endif
```

```cpp
// ActorManager.cpp
#include "ActorManager.h"

// アクターの追加
void ActorManager::add(const ActorPtr& actor) {
    actors_.push_front(actor);
}
// 更新
void ActorManager::update(float delta_time) {
    for (const auto& actor : actors_) {
        actor->update(delta_time);
    }
}
// 描画
void ActorManager::draw() const {
    for (const auto& actor : actors_) {
        actor->draw();
    }
}
// 衝突判定
void ActorManager::collide() {
    for (auto i = actors_.begin(); i != actors_.end(); ++i) {
        for (auto j = std::next(i); j != actors_.end(); ++j) {
            (*i)->collide(**j);
        }
    }
}
// 衝突判定
void ActorManager::collide(Actor& other) {
    for (const auto& actor : actors_) {
        other.collide(*actor);
    }
}
// 衝突判定
void ActorManager::collide(ActorManager &other) {
    for (const auto& actor : actors_) {
        other.collide(*actor);
    }
}
// 削除
void ActorManager::remove() {
    actors_.remove_if([](const ActorPtr& actor) { return actor->is_dead(); });
}
// アクターの検索
ActorPtr ActorManager::find(const std::string& name) const {
    for (const auto& actor : actors_) {
        if (actor->name() == name) {
            return actor;
        }
    }
    return ActorPtr();
}
// メッセージ処理
void ActorManager::handle_message(EventMessage message, void* param) {
    for (const auto& actor : actors_) {
        actor->handle_message(message, param);
    }
}
// アクター数の取得
unsigned int ActorManager::count() const {
    return actors_.size();
}
// 消去
void ActorManager::clear() {
    actors_.clear();
}
```

■アクターグループ列挙型

```
// ActorGrop.h
#ifndef ACTOR_GROUP_H_
#define ACTOR_GROUP_H_

// アクターグループ
enum class ActorGroup {
    Player,         // プレイヤー
    Enemy,          // 敵
    PlayerBullet,   // プレイヤー弾
    EnemyBullet,    // 敵弾
    Effect          // エフェクト
};

#endif
```

■アクターグループマネージャー

```
// ActorGropManager.h
#ifndef ACTOR_GROUP_MANAGER_H_
#define ACTOR_GROUP_MANAGER_H_

#include "ActorManager.h"
#include <map>

enum class ActorGroup;

// アクターグループマネージャ
class ActorGroupManager {
public:
    // コンストラクタ
    ActorGroupManager() = default;
    // グループの追加
    void add(ActorGroup group);
    // アクターの追加
    void add(ActorGroup group, const ActorPtr& actor);
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // 描画
    void draw(ActorGroup group) const;
    // 消去
    void clear();
    // アクターを検索
    ActorPtr find(ActorGroup group, const std::string& name) const;
    // アクター数を返す
    unsigned int count(ActorGroup group) const;
    // 衝突判定
    void collide(ActorGroup group1, ActorGroup group2);
    // 削除
    void remove();
    // メッセージ処理
    void handle_message(EventMessage message, void* param);
    // コピー禁止
    ActorGroupManager(const ActorGroupManager& other) = delete;
    ActorGroupManager& operator=(const ActorGroupManager& other) = delete;

private:
    using ActorGroupMap = std::map<ActorGroup, ActorManager>;
    ActorGroupMap   actor_group_map_;
};

#endif
```

```cpp
// ActorGropManager.cpp
#include "ActorGroupManager.h"

// グループの追加
void ActorGroupManager::add(ActorGroup group) {
    actor_group_map_[group].clear();
}

// アクターの追加
void ActorGroupManager::add(ActorGroup group, const ActorPtr& actor) {
    actor_group_map_[group].add(actor);
}

// 更新
void ActorGroupManager::update(float delta_time) {
    for (auto& pair : actor_group_map_) {
        pair.second.update(delta_time);
    }
}

// 描画
void ActorGroupManager::draw() const {
    for (auto& pair : actor_group_map_) {
        draw(pair.first);
    }
}

// 描画
void ActorGroupManager::draw(ActorGroup group) const {
    actor_group_map_.at(group).draw();
}

// 消去
void ActorGroupManager::clear() {
    actor_group_map_.clear();
}

// アクターの検索
ActorPtr ActorGroupManager::find(ActorGroup group, const std::string& name) const {
    return actor_group_map_.at(group).find(name);
}

// アクター数を返す
unsigned int ActorGroupManager::count(ActorGroup group) const {
    return actor_group_map_.at(group).count();
}

// 衝突判定
void ActorGroupManager::collide(ActorGroup group1, ActorGroup group2) {
    actor_group_map_[group1].collide(actor_group_map_[group2]);
}

// 削除
void ActorGroupManager::remove() {
    for (auto& pair : actor_group_map_) {
        pair.second.remove();
    }
}

// メッセージ処理
void ActorGroupManager::handle_message(EventMessage message, void* param){
    for (auto& pair : actor_group_map_) {
        pair.second.handle_message(message, param);
    }
}
```

■テクスチャ ID

```
// TextureID.h
#ifndef TEXTURE_ID_H_
#define TEXTURE_ID_H_

enum {
    TEXTURE_BG1,
    TEXTURE_BG2,
    TEXTURE_BG3,
    TEXTURE_SHIP,
    TEXTURE_ENEMY,
    TEXTURE_ENEMY2,
    TEXTURE_BEAM,
    TEXTURE_EBEAM,
    TEXTURE_BOMB,
    TEXTURE_NUMBER
};

#endif
```

■フィールドクラス

```
// Field.h
#ifndef FIELD_H_
#define FIELD_H_

#include "BoundingRectangle.h"

// フィールドクラス
class Field {
public:
    // コンストラクタ
    Field() = default;
    // コンストラクタ
    explicit Field(const BoundingRectangle& area);
    // 初期化
    void initialize();
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // エリアの外側か？
    bool is_outside(const BoundingRectangle& rect);
    // エリアの取得
    const BoundingRectangle& area() const;

private:
    // 背景描画
    void draw_background(GSuint id, float scroll) const;

private:
    // ゲーム画面エリア
    BoundingRectangle area_ { 0.0f, 0.0f, 640.0f, 480.0f };
    // 拡張エリア
    BoundingRectangle extend_area_ { area_.expand({ 128.0f, 128.0f }) };
    // スクロール位置
    float scroll_{ 0.0f };
};

#endif
```

```cpp
// Field.cpp
#include "Field.h"
#include "TextureID.h"
#include <cmath>

// コンストラクタ
Field::Field(const BoundingRectangle& area) :
    area_{ area },
    extend_area_{ area.expand({ 128.0f, 128.0 }) } {
}

// 初期化
void Field::initialize() {
    scroll_ = 0.0f;
}

// 更新
void Field::update(float delta_time) {
    scroll_ += delta_time;
}

// 描画
void Field::draw() const {
    draw_background(TEXTURE_BG3, scroll_ * 0.5f);
    draw_background(TEXTURE_BG2, scroll_ * 2.0f);
    draw_background(TEXTURE_BG1, scroll_ * 3.0f);
}

// エリアの外側か？
bool Field::is_outside(const BoundingRectangle& rect) {
    return !extend_area_.intersects(rect);
}

// エリアの取得
const BoundingRectangle& Field::area() const {
    return area_;
}

// 背景描画
void Field::draw_background(GSuint id, float scroll) const {
    float left  = std::fmod(scroll, area_.max().x);
    float width = area_.max().x - left;
    float right = left + width;
    GSvector2 posl{ 0.0f, 0.0f };
    GSrect rect;
    rect.left   = left;
    rect.top    = 0.0f;
    rect.right  = right;
    rect.bottom = area_.max().y;
    gsDrawSprite2D(id, &posl, &rect, NULL, NULL, NULL, 0.0f);
    GSvector2 posr{ width, 0.0f };
    rect.left  = 0.0f;
    rect.right = left;
    gsDrawSprite2D(id, &posr, &rect, NULL, NULL, NULL, 0.0f);
}
```

■ワールド抽象インターフェース

```
// IWorld.h
#ifndef IWORLD_H_
#define IWORLD_H_

#include "ActorPtr.h"
#include <string>

enum class ActorGroup;
enum class EventMessage;
class Field;

// ワールド抽象インターフェース
class IWorld {
public:
    // 仮想デストラクタ
    virtual ~IWorld() {}
    // アクターを追加
    virtual void add_actor(ActorGroup group, const ActorPtr& actor) = 0;
    // アクターを検索
    virtual ActorPtr find_actor(ActorGroup group, const std::string& name) const = 0;
    // アクター数の取得
    virtual unsigned int count_actor(ActorGroup group) const = 0;
    // メッセージを送信
    virtual void send_message(EventMessage message, void* param = nullptr) = 0;
    // フィールドを取得
    virtual Field& field() = 0;
};

#endif
```

■ワールドクラス

```
// World.h
#ifndef WORLD_H_
#define WORLD_H_

#include "IWorld.h"
#include "ActorGroupManager.h"
#include "Field.h"
#include "ActorPtr.h"
#include <functional>

enum class EventMessage;

// ワールドクラス
class World : public IWorld {
public:
    // イベントメッセージリスナー
    using EventMessageListener = std::function<void(EventMessage, void*)>;

public:
    // コンストラクタ
    World();
    // 初期化
    void initialize();
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // メッセージ処理
    void handle_message(EventMessage message, void* param);
    // メッセージリスナーの登録
    void add_event_message_listener(EventMessageListener listener);
    // 消去
    void clear();

    // アクターの追加
    virtual void add_actor(ActorGroup group, const ActorPtr& actor) override;
    // アクターの検索
    virtual ActorPtr find_actor(ActorGroup group, const std::string& name) const override;
    // アクター数の取得
    virtual unsigned int count_actor(ActorGroup group) const override;
    // メッセージの送信
    virtual void send_message(EventMessage message, void* param = nullptr) override;
    // フィールドを取得
    virtual Field& field() override;

    // コピー禁止
    World(const World& other) = delete;
    World& operator = (const World& other) = delete;

private:
    // アクターマネージャー
    ActorGroupManager actors_;
    // フィールド
    Field field_;
    // メッセージリスナ
    EventMessageListener listener_ { [](EventMessage, void*) {} };
};

#endif
```

```cpp
// World.cpp
#include "World.h"
#include "Actor.h"
#include "ActorGroup.h"

// コンストラクタ
World::World() {
    initialize();
}
// 初期化
void World::initialize() {
    clear();
    actors_.add(ActorGroup::Player);
    actors_.add(ActorGroup::PlayerBullet);
    actors_.add(ActorGroup::Enemy);
    actors_.add(ActorGroup::EnemyBullet);
    actors_.add(ActorGroup::Effect);
    field_.initialize();
}
// 更新
void World::update(float delta_time) {
    field_.update(delta_time);
    actors_.update(delta_time);
    actors_.collide(ActorGroup::Player, ActorGroup::Enemy);
    actors_.collide(ActorGroup::Player, ActorGroup::EnemyBullet);
    actors_.collide(ActorGroup::PlayerBullet, ActorGroup::Enemy);
    actors_.remove();
}
// 描画
void World::draw() const {
    field_.draw();
    actors_.draw();
}
// メッセージ処理
void World::handle_message(EventMessage message, void* param) {
    listener_(message, param);
    actors_.handle_message(message, param);
}
// イベントメッセージリスナーの登録
void World::add_event_message_listener(EventMessageListener listener) {
    listener_ = listener;
}
// 消去
void World::clear() {
    actors_.clear();
    listener_ = [](EventMessage, void*) {};
}
// アクターの追加
void World::add_actor(ActorGroup group, const ActorPtr& actor) {
    actors_.add(group, actor);
}
// アクターの検索
ActorPtr World::find_actor(ActorGroup group, const std::string& name) const {
    return actors_.find(group, name);
}
// アクター数を返す
unsigned int World::count_actor(ActorGroup group) const {
    return actors_.count(group);
}
// メッセージの送信
void World::send_message(EventMessage message, void* param) {
    handle_message(message, param);
}
// フィールドを取得
Field& World::field() {
    return field_;
}
```

■プレーヤクラス

```
// Player.h
#ifndef PLAYER_H_
#define PLAYER_H_

#include "Actor.h"

// プレーヤ
class Player : public Actor {
public:
    // コンストラクタ
    Player(IWorld& world, const GSvector2& position);
    // 更新
    virtual void update(float delta_time) override;
    // 衝突リアクション
    virtual void react(Actor& other) override;
};

#endif

// Player.cpp
#include "Player.h"
#include "IWorld.h"
#include "ActorGroup.h"
#include "PlayerBeam.h"
#include "Explosion.h"
#include "TextureID.h"
#include "Field.h"

// コンストラクタ
Player::Player(IWorld& world, const GSvector2& position) {
    world_    = &world;
    name_     = "Player";
    position_ = position;
    body_     = BoundingRectangle{ 0.0f, 0.0f, 64.0f, 40.0f };
    texture_  = TEXTURE_SHIP;
}

// 更新
void Player::update(float delta_time) {
    GSvector2 velocity{ 0.0f, 0.0f };
    if (gsGetKeyState(GKEY_LEFT))  velocity.x = -1.0f;
    if (gsGetKeyState(GKEY_RIGHT)) velocity.x =  1.0f;
    if (gsGetKeyState(GKEY_UP))    velocity.y = -1.0f;
    if (gsGetKeyState(GKEY_DOWN))  velocity.y =  1.0f;
    velocity_ = velocity.getNormalized() * 8.0f;
    position_ += velocity_ * delta_time;
    const auto& area = world_->field().area();
    position_ = position_.clamp(area.min(), area.max() - body_.max());
    if (gsGetKeyTrigger(GKEY_SPACE)) {
        world_->add_actor(ActorGroup::PlayerBullet,
            new_actor<PlayerBeam>(*world_, position_ + GSvector2{ 50.0f, 25.0f }, GSvector2{ 4.0f, 0.0f }));
    }
}

// 衝突リアクション
void Player::react(Actor&) {
    world_->add_actor(ActorGroup::Effect, new_actor<Explosion>(*world_, position_));
    die();
}
```

■プレーヤ弾クラス

```
// PlayerBeam.h
#ifndef PLAYER_BEAM_H_
#define PLAYER_BEAM_H_

#include "Actor.h"

// プレーヤ弾
class PlayerBeam : public Actor {
public:
    // コンストラクタ
    PlayerBeam(IWorld& world, const GSvector2& position, const GSvector2& velocity);
    // 更新
    virtual void update(float time) override;
    // 衝突リアクション
    virtual void react(Actor& other) override;
};

#endif

// PlayerBeam.cpp
#include "PlayerBeam.h"
#include "Field.h"
#include "IWorld.h"
#include "TextureID.h"

// コンストラクタ
PlayerBeam::PlayerBeam(IWorld& world, const GSvector2& position, const GSvector2& velocity) {
    world_    = &world;
    name_     = "PlayerBeam";
    position_ = position;
    velocity_ = velocity;
    body_     = BoundingRectangle{ 0.0f, 0.0f, 16.0f, 8.0f };
    texture_  = TEXTURE_BEAM;
}

// 更新
void PlayerBeam::update(float delta_time) {
    position_ += velocity_ * delta_time;
    if (world_->field().is_outside(body())) {
        die();
    }
}

// 衝突リアクション
void PlayerBeam::react(Actor&) {
    die();
}
```

■敵青クラス

```
// EnemyBlue.h
#ifndef ENEMY_BLUE_H_
#define ENEMY_BLUE_H_

#include "Actor.h"

// 敵クラス
class EnemyBlue : public Actor {
public:
    // コンストラクタ
    EnemyBlue(IWorld& world, const GSvector2& position);
    // 更新
    virtual void update(float delta_time) override;
    // 衝突リアクション
    void react(Actor & other);

private:
    float timer_{ 0.0f };
};

#endif

// EnemyBlue.cpp
#include "EnemyBlue.h"
#include "IWorld.h"
#include "Field.h"
#include "EnemyBeam.h"
#include "Explosion.h"
#include "ActorGroup.h"
#include "TextureID.h"

// コンストラクタ
EnemyBlue::EnemyBlue(IWorld& world, const GSvector2& position) {
    world_    = &world;
    name_     = "EnemyBlue";
    position_ = position;
    velocity_ = GSvector2{ -1.5f, 0.0f };
    body_     = BoundingRectangle{ 0.0f, 0.0f, 32.0f, 32.0f };
    texture_  = TEXTURE_ENEMY2;
}

// 更新
void EnemyBlue::update(float delta_time) {
    position_ += velocity_ * delta_time;
    timer_ += delta_time;
    if (timer_ > 120.0f) {
        const auto player = world_->find_actor(ActorGroup::Player, "Player");
        if (player != nullptr) {
            const auto velocity = (player->position() - position_).getNormalized() * 4.0f;
            world_->add_actor(ActorGroup::EnemyBullet,
                new_actor<EnemyBeam>(*world_, position_, velocity));
        }
        timer_ = 0.0f;
    }
    if (world_->field().is_outside(body())) {
        die();
    }
}

// 衝突リアクション
void EnemyBlue::react(Actor&) {
    world_->add_actor(ActorGroup::Effect, new_actor<Explosion>(*world_, position_));
    die();
}
```

■敵赤クラス

```
// EnemyRed.h
#ifndef ENEMY_RED_H_
#define ENEMY_RED_H_

#include "Actor.h"

// 敵クラス
class EnemyRed : public Actor {
public:
    // コンストラクタ
    EnemyRed(IWorld& world, const GSvector2& position);
    // 更新
    virtual void update(float delta_time) override;
    // 衝突リアクション
    virtual void react(Actor& other) override;
private:
    float timer_{ 0.0f };
};

#endif

// EnemyRed.cpp
#include "EnemyRed.h"
#include "IWorld.h"
#include "Field.h"
#include "EnemyBeam.h"
#include "Explosion.h"
#include "ActorGroup.h"
#include "TextureID.h"

// コンストラクタ
EnemyRed::EnemyRed(IWorld& world, const GSvector2& position) {
    world_    = &world;
    name_     = "EnemyRed";
    position_ = position;
    velocity_ = GSvector2{ -2.0f, 2.0f };
    body_     = BoundingRectangle{ 0.0f, 0.0f, 32.0f, 32.0f };
    texture_  = TEXTURE_ENEMY;
}

// 更新
void EnemyRed::update(float delta_time)  {
    if (position_.y < world_->field().area().min().y) {
        velocity_.y = 2.0f;
    }
    if (position_.y > (world_->field().area().max().y - body_.max().y)) {
        velocity_.y = -2.0f;
    }
    position_ += velocity_ * delta_time;
    if (timer_ > 60.0f) {
        world_->add_actor(ActorGroup::EnemyBullet,
            new_actor<EnemyBeam>(*world_, position_, GSvector2{ -4.0f, 0.0f }));
        timer_ = 0.0f;
    }
    timer_ += delta_time;
    if (world_->field().is_outside(body())) {
        die();
    }
}

// 衝突リアクション
void EnemyRed::react(Actor&) {
    world_->add_actor(ActorGroup::Effect, new_actor<Explosion>(*world_, position_));
    die();
}
```

■敵弾クラス

```
// EnemyBeam.h
#ifndef ENEMY_BEAM_H_
#define ENEMY_BEAM_H_

#include "Actor.h"

// 敵弾クラス
class EnemyBeam : public Actor {
public:
    // コンストラクタ
    EnemyBeam(IWorld& world, const GSvector2& position, const GSvector2& velocity);
    // 更新
    virtual void update(float delta_time) override;
    // 衝突リアクション
    virtual void react(Actor& other) override;
};

#endif

// EnemyBeam.cpp
#include "EnemyBeam.h"
#include "IWorld.h"
#include "Field.h"
#include "TextureID.h"

// コンストラクタ
EnemyBeam::EnemyBeam(IWorld& world, const GSvector2& position, const GSvector2& velocity) {
    world_    = &world;
    name_     = "EnemyBeam";
    position_ = position;
    velocity_ = velocity;
    body_     = BoundingRectangle{ 0.0f, 0.0f, 16.0f, 16.0f };
    texture_  = TEXTURE_EBEAM;
}

// 更新
void EnemyBeam::update(float delta_time) {
    position_ += velocity_ * delta_time;
    if (world_->field().is_outside(body())) {
        die();
    }
}

// 衝突リアクション
void EnemyBeam::react(Actor&) {
    die();
}
```

■爆発クラス

```
// Explosion.h
#ifndef EXPLOSION_H_
#define EXPLOSION_H_

#include "Actor.h"

// 爆発クラス
class Explosion : public Actor {
public:
    // コンストラクタ
    Explosion(IWorld& world, const GSvector2& position);
    // 更新
    virtual void update(float time) override;

private:
    float timer_{ 0.0f };
};

#endif

// Explosion.cpp
#include "Explosion.h"
#include "TextureID.h"

// コンストラクタ
Explosion::Explosion(IWorld& world, const GSvector2& position) {
    world_    = &world;
    name_     = "Explosion";
    position_ = position;
    texture_  = TEXTURE_BOMB;
}

// 更新
void Explosion::update(float delta_time)  {
    if (timer_ > 30.0f) {
        die();
    }
    timer_ += delta_time;
}
```

■敵生成クラス

```
// EnemyGenerator.h
#ifndef ENEMY_GENERATOR_H_
#define ENEMY_GENERATOR_H_

#include "Actor.h"

// 敵生成クラス
class EnemyGenerator : public Actor {
public:
    // コンストラクタ
    explicit EnemyGenerator(IWorld& world);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override {}
private:
    float timer_{ 0.0f };
};

#endif

// EnemyGenerator.cpp
#include "EnemyGenerator.h"
#include "IWorld.h"
#include "Field.h"
#include "ActorGroup.h"
#include "EnemyRed.h"
#include "EnemyBlue.h"

// コンストラクタ
EnemyGenerator::EnemyGenerator(IWorld& world) {
    world_ = &world;
    name_  = "EnemyGenerator";
}

// 更新
void EnemyGenerator::update(float delta_time) {
    if (timer_ < 0.0f) {
        const GSvector2 position{ world_->field().area().max().x + 64.0f, gsRandf(0.0f, world_->field().area().max().y) };
        if (gsRand(0, 1) == 0) {
            world_->add_actor(ActorGroup::Enemy, new_actor<EnemyRed>(*world_, position));
        } else {
            world_->add_actor(ActorGroup::Enemy, new_actor<EnemyBlue>(*world_, position));
        }
        timer_ = gsRandf(30.0f, 120.0f);
    }
    timer_ -= delta_time;
}
```

■シューティングアプリケーションクラス

```
// ShootingApplication.h
#ifndef SHOOTING_APPLICATION_H_
#define SHOOTING_APPLICATION_H_

#include <GSgame.h>
#include "World.h"

class ShootingApplication : public gslib::Game {
private:
    // 開始
    virtual void start() override;
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() override;
    // 終了
    virtual void end() override;
private:
    World world_;   // ワールド
};

#endif

// ShootingApplication.cpp
#include "ShootingApplication.h"
#include "Player.h"
#include "Explosion.h"
#include "EnemyGenerator.h"
#include "ActorGroup.h"
#include "TextureID.h"

// 開始
void ShootingApplication::start() {
    gsLoadTexture(TEXTURE_BG3,     "asset/BG3.BMP");
    gsLoadTexture(TEXTURE_SHIP,    "asset/SHIP.BMP");
    gsLoadTexture(TEXTURE_BEAM,    "asset/BEAM.BMP");
    gsLoadTexture(TEXTURE_ENEMY,   "asset/ENEMY.BMP");
    gsLoadTexture(TEXTURE_ENEMY2, "asset/ENEMY2.BMP");
    gsLoadTexture(TEXTURE_EBEAM,   "asset/EBEAM.BMP");
    gsLoadTexture(TEXTURE_BG1,     "asset/BG1.BMP");
    gsLoadTexture(TEXTURE_BG2,     "asset/BG2.BMP");
    gsLoadTexture(TEXTURE_BOMB,    "asset/BOMB.BMP");
    world_.add_actor(ActorGroup::Player, new_actor<Player>(world_, GSvector2{ 100.0f, 240.0f }));
    world_.add_actor(ActorGroup::Effect, new_actor<EnemyGenerator>(world_));
}

// 更新
void ShootingApplication::update(float delta_time) {
    world_.update(delta_time);
}

// 描画
void ShootingApplication::draw() {
    world_.draw();
}

// 終了
void ShootingApplication::end() {
    world_.clear();
}
```

■メイン関数

```
// main.cpp
#include "ShootingApplication.h"

int main() {
    return ShootingApplication().run();
}
```

■ナンバーテクスチャクラス

```
// NumberTexture.h
#ifndef NUMBER_TEXTURE_H_
#define NUMBER_TEXTURE_H_

#include <gslib.h>
#include <string>

// ナンバテクスチャ
class NumberTexture {
public:
    // コンストラクタ
    NumberTexture(GSint texture, int width, int height);
    // 描画
    void draw(const GSvector2& position, int num, int digit, char fill = '0') const;
    // 描画
    void draw(const GSvector2& position, int num) const;
    // 描画
    void draw(const GSvector2& position, const std::string& num) const;

private:
    // フォント用のテクスチャ
    GSint   texture_;
    // 文字の幅
    int     width_;
    // 文字の高さ
    int     height_;
};

#endif

// NumberTexture.cpp
#include "NumberTexture.h"
#include <sstream>
#include <iomanip>

// コンストラクタ
NumberTexture::NumberTexture(GSint texture, int width, int height) :
    texture_{ texture }, width_{ width }, height_{ height } {
}

// 描画
void NumberTexture::draw(const GSvector2& position, int num, int digit, char fill) const {
    std::stringstream ss;
    ss << std::setw(digit) << std::setfill(fill) << num;
    draw(position, ss.str());
}

// 描画
void NumberTexture::draw(const GSvector2& position, int num) const {
    draw(position, std::to_string(num));
}

// 描画
void NumberTexture::draw(const GSvector2& position, const std::string& num) const {
    for (int i = 0; i < (int)num.size(); ++i) {
        if (num[i] == ' ') continue;
        const int n = num[i] - '0';
        const GSrect rect(n * width_, 0.0f, (n * width_) + width_, height_);
        const GSvector2 pos{ position.x + i * width_, position.y };
        gsDrawSprite2D(texture_, &pos, &rect, NULL, NULL, NULL, 0);
    }
}
```

■スコアクラスの作成

```
// Score.h
#ifndef SCORE_H_
#define SCORE_H_

// スコアクラス
class Score {
public:
    // コンストラクタ
    Score(int score = 0);
    // スコアの初期化
    void initialize(int score = 0);
    // スコアの加算
    void add(int score);
    // スコアの描画
    void draw() const;
    // スコアの取得
    int get() const;

private:
    // スコア
    int score_;
};

#endif

// Score.cpp
#include "Score.h"
#include "NumberTexture.h"
#include "TextureID.h"
#include <gslib.h>
#include <algorithm>

// コンストラクタ
Score::Score(int score) :
    score_{ score } {
}

// スコアの初期化
void Score:: initialize(int score) {
    score_ = score;
}

// スコアの加算
void Score::add(int score) {
    score_ = std::min(score_ + score, 9999999);
}

// スコアの描画
void Score::draw() const {
    static const NumberTexture number{ TEXTURE_NUMBER, 16, 16 };
    number.draw(GSvector2{0, 0}, score_, 7);
}

// スコアの取得
int Score::get() const {
    return score_;
}
```

■シーン抽象インターフェース

```
// IScene.h
#ifndef ISCENE_H_
#define ISCENE_H_

enum class Scene;

// シーン抽象インターフェース
class IScene {
public:
    // 仮想デストラクタ
    virtual ~IScene() {}
    // 開始
    virtual void start() = 0;
    // 更新
    virtual void update(float delta_time) = 0;
    // 描画
    virtual void draw() const = 0;
    // 終了しているか？
    virtual bool is_end() const = 0;
    // 次のシーンを返す
    virtual Scene next() const = 0;
    // 終了
    virtual void end() = 0;
};

#endif
```

■シーンポインタ

```
// IScenePtr.h
#ifndef ISCENE_PTR_H_
#define ISCENE_PTR_H_

#include <memory>

// シーンポインタ
class IScene;
using IScenePtr = std::shared_ptr<IScene>;

// シーンの作成
template<class T, class... Args>
inline IScenePtr new_scene(Args&&... args) {
    return std::make_shared<T>(args...);
}

#endif
```

■シーン名列挙型

```
// Scene.h
#ifndef SCENE_H_
#define SCENE_H_

// シーン名
enum class Scene {
    None,       // ダミー
    Title,      // タイトル
    GamePlay,   // ゲーム中
    GameOver,   // ゲームオーバー
    Ending      // エンディング
};

#endif
```

■ ヌルシーンクラス

```
// NullScene.h
#ifndef SCENE_NULL_H_
#define SCENE_NULL_H_

#include "IScene.h"

// 空のシーン
class SceneNull : public IScene {
public:
    // 開始
    virtual void start() override;
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const  override;
    // 終了しているか？
    virtual bool is_end () const  override;
    // 次のシーンを返す
    virtual Scene next() const  override;
    // 終了
    virtual void end()  override;
};

#endif

// NullScene.cpp
#include "SceneNull.h"
#include "Scene.h"

// 開始
void SceneNull::start() {}

// 更新
void SceneNull::update(float) {}

// 描画
void SceneNull::draw() const {}

// 終了しているか？
bool SceneNull::is_end() const {
    return false;
}

// 次のシーンを返す
Scene SceneNull::next() const {
    return Scene::None;
}

// 終了
void SceneNull::end() {}
```

■ シーンマネージャクラス

```
// SceneManager.h
#ifndef SCENE_MANAGER_H_
#define SCENE_MANAGER_H_

#include "IScenePtr.h"
#include <unordered_map>

enum class Scene;

// シーン管理クラス
class SceneManager {
public:
    // コンストラクタ
    SceneManager();
    // 初期化
    void initialize();
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // 終了
    void end();
    // シーンの追加
    void add(Scene name, const IScenePtr& scene);
    // シーンの変更
    void change(Scene name);
    // コピー禁止
    SceneManager(const SceneManager& other) = delete;
    SceneManager& operator = (const SceneManager& other) = delete;
private:
    // シーン
    std::unordered_map<Scene, IScenePtr> scenes_;
    // 現在のシーン
    IScenePtr current_scene_;
};

#endif
```

```
// SceneManager.cpp
#include "SceneManager.h"
#include "SceneNull.h"

// コンストラクタ
SceneManager::SceneManager() :
    current_scene_(new_scene<SceneNull>()) {
}

// 初期化
void SceneManager::initialize() {
    end();
    scenes_.clear();
}

// 更新
void SceneManager::update(float delta_time) {
    current_scene_->update(delta_time);
    if (current_scene_->is_end()) {
        change(current_scene_->next());
    }
}

// 描画
void SceneManager::draw() const {
    current_scene_->draw();
}

// 終了
void SceneManager::end() {
    current_scene_->end();
    current_scene_ = new_scene<SceneNull>();
}

// シーンの追加
void SceneManager::add(Scene name, const IScenePtr& scene) {
    scenes_[name] = scene;
}

// シーンの変更
void SceneManager::change(Scene name) {
    end();
    current_scene_ = scenes_[name];
    current_scene_->start();
}
```

■ タイトルシーンクラス

```
// TitleScene.h
#ifndef TITLE_SCENE_H_
#define TITLE_SCENE_H_

#include "IScene.h"

// タイトルシーン
class TitleScene : public IScene {
public:
    // コンストラクタ
    TitleScene() = default;
    // 開始
    virtual void start() override;
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
    // 終了しているか？
    virtual bool is_end() const override;
    // 次のシーンを返す
    virtual Scene next() const override;
    // 終了
    virtual void end() override;
private:
    bool        is_end_{ false };   // 終了フラグ
};

#endif

// TitleScene.cpp
#include "TitleScene.h"
#include "Scene.h"
#include <gslib.h>

// 開始
void TitleScene::start() {
    is_end_ = false;
}

// 更新
void TitleScene::update(float) {
    if (gsGetKeyTrigger(GKEY_SPACE) == GS_TRUE) {
        is_end_ = true;
    }
}

// 描画
void TitleScene::draw() const {
    gsDrawText("タイトル画面");
}

// 終了しているか？
bool TitleScene::is_end() const {
    return is_end_;
}

// 次のシーンを返す
Scene TitleScene::next() const {
    return Scene::GamePlay;
}

// 終了
void TitleScene::end() {}
```

■ イベントメッセージ

```
// EventMessage.h
#ifndef EVENT_MESSAGE_H_
#define EVENT_MESSAGE_H_

// メッセージ
enum class EventMessage {
    None,           // ダミー
    AddScore        // 得点を加算
};

#endif
```

■ ゲームプレイシーンクラス

```
// GamePlayScene.h
#ifndef GAME_PLAY_SCENE_H_
#define GAME_PLAY_SCENE_H_

#include "IScene.h"
#include "World.h"
#include "Score.h"

// ゲームプレイシーンクラス
class GamePlayScene : public IScene {
public:
    // コンストラクタ
    GamePlayScene() = default;
    // 開始
    virtual void start() override;
    // 更新
    virtual void update(float deltaTime) override;
    // 描画
    virtual void draw() const override;
    // 終了しているか？
    virtual bool is_end() const  override;
    // 次のシーンを返す
    virtual Scene next() const  override;
    // 終了
    virtual void end() override;
    // メッセージ処理
    void handle_message(EventMessage message, void * param);
private:
    World       world_;             // ワールドクラス
    Score       score_;             // スコアクラス
    bool        is_end_{ false };   // 終了フラグ
};

#endif
```

```
// GamePlayScene.cpp
#include "Player.h"
#include "EnemyGenerator.h"
#include "ActorGroup.h"
#include "EventMessage.h"
#include "Scene.h"
#include "TextureID.h"

// 開始
void GamePlayScene::start() {
    is_end_ = false;
    gsLoadTexture(TEXTURE_BG3,     "asset/BG3.BMP");
    gsLoadTexture(TEXTURE_SHIP,    "asset/SHIP.BMP");
    gsLoadTexture(TEXTURE_BEAM,    "asset/BEAM.BMP");
    gsLoadTexture(TEXTURE_ENEMY,   "asset/ENEMY.BMP");
    gsLoadTexture(TEXTURE_ENEMY2,  "asset/ENEMY2.BMP");
    gsLoadTexture(TEXTURE_EBEAM,   "asset/EBEAM.BMP");
    gsLoadTexture(TEXTURE_BG1,     "asset/BG1.BMP");
    gsLoadTexture(TEXTURE_BG2,     "asset/BG2.BMP");
    gsLoadTexture(TEXTURE_BOMB,    "asset/BOMB.BMP");
    gsLoadTexture(TEXTURE_NUMBER,  "asset/NUM.BMP");
    score_.initialize();
    world_.initialize();
    // イベントメッセージの登録
    world_.add_event_message_listener([=](EventMessage msg, void* param) { handle_message(msg, param); } );
    world_.add_actor(ActorGroup::Player, new_actor<Player>(world_, GSvector2{ 100.0f, 240.0f }));
    world_.add_actor(ActorGroup::Effect, new_actor<EnemyGenerator>(world_));
}
// 更新
void GamePlayScene::update(float delta_time) {
    world_.update(delta_time);
}
// 描画
void GamePlayScene::draw() {
    world_.draw();
    score_.draw();
}
// 終了しているか？
bool GamePlayScene::is_end() const {
    return is_end_;
}
// 次のシーンを返す
Scene GamePlayScene::next() const {
    return Scene::Title;
}
// 終了
void GamePlayScene::end() {
    world_.clear();
    gsDeleteTexture(TEXTURE_BG3);
    gsDeleteTexture(TEXTURE_SHIP);
    gsDeleteTexture(TEXTURE_BEAM);
    gsDeleteTexture(TEXTURE_ENEMY);
    gsDeleteTexture(TEXTURE_ENEMY2);
    gsDeleteTexture(TEXTURE_EBEAM);
    gsDeleteTexture(TEXTURE_BG1);
    gsDeleteTexture(TEXTURE_BG2);
    gsDeleteTexture(TEXTURE_BOMB);
    gsDeleteTexture(TEXTURE_NUMBER);
}
// メッセージ処理
void GamePlayScene::handle_message(EventMessage message, void * param) {
    switch (message) {
    case EventMessage::AddScore:
        int* score = (int*)param;
        score_.add(*score);
        break;
    }
}
```

■ シューティングアプリケーションクラスの変更

```
// ShootingApplication.h
#ifndef SHOOTING_APPLICATION_H_
#define SHOOTING_APPLICATION_H_

#include <GSgame.h>
#include "SceneManager.h"

class ShootingApplication : public gslib::Game {
private:
    // 開始
    virtual void start() override;
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() override;
    // 終了
    virtual void end() override;

private:
    // シーンマネージャー
    SceneManager scene_manager_;
};

#endif

// ShootingApplication.cpp
#include "ShootingApplication.h"
#include "SceneManager.h"
#include "Scene.h"
#include "TitleScene.h"
#include "GamePlayScene.h"

// 開始
void ShootingApplication::start() {
    scene_manager_.add(Scene::Title, new_scene<TitleScene>());
    scene_manager_.add(Scene::GamePlay, new_scene<GamePlayScene>());
    scene_manager_.change(Scene::Title);
}

// 更新
void ShootingApplication::update(float deltaTime) {
    scene_manager_.update(deltaTime);
}

// 描画
void ShootingApplication::draw() {
    scene_manager_.draw();
}

// 終了
void ShootingApplication::end() {
    scene_manager_.end();
}
```